UNITED STATES PATENT APPLICATION

FOR

EFFICIENT LOADING OF DATA INTO A RELATIONAL DATABASE

INVENTORS:

NAMIT JAIN
FOLSOM, CALIFORNIA
CITIZENSHIP: INDIA

NIPUN AGARWAL
SANTA CLARA, CALIFORNIA
CITIZENSHIP: INDIA

RAVI MURTHY
FOSTER CITY, CALIFORNIA
CITIZENSHIP: INDIA

ASSIGNEE:

ORACLE INTERNATIONAL CORPORATION
500 ORACLE PARKWAY
REDWOOD SHORES, CA 94065

PREPARED BY:
PETER MEI
BINGHAM MCCUTCHEN LLP
THREE EMBARCADERO CENTER, SUITE 1800
SAN FRANCISCO, CA 94111

EXPRESSMAIL NO. EV 313810739 US

EFFICIENT LOADING OF DATA INTO A RELATIONAL DATABASE

RELATED APPLICATIONS

[0001]   The present application is a continuation-in-part of U.S. Patent Application Serial

No. **10/648,577** , entitled DIRECT LOADING OF OPAQUE TYPES, (Attorney Docket No.

5   50277-2236) and U.S. Patent Application Serial No. **10/648,600**, entitled DIRECT

LOADING OF SEMISTRUCTURED DATA, by Namit Jain, Nipun Agarwal, and Ravi

Murthy (Attorney Docket No. 50277-2235), both filed on August 25, 2003, which are hereby

incorporated by reference in their entirety.

[0002]   The present application is also related to the following U.S. Patent Applications,

10   the entire contents of which are incorporated herein by reference for all purposes:

[0003]   U.S. Patent Application Serial No. 10/192,411, filed on July 9, 2002, entitled

OPAQUE TYPES, by Rajagopalan Govindarajan, Viswanathan Krishnamurthy, and Anil

Nori;

[0004]   U.S. Patent Application Serial No. 10/260,138, filed on September 27, 2002,

15   entitled OPERATORS FOR ACCESSING HIERARCHICAL DATA IN A RELATIONAL

SYSTEM, by Nipun Agarwal, Ravi Murthy, Eric Sedlar, Sivasankaran Chandrasekar and Fei

Ge;

[0005]   U.S. Patent Application Serial No. 10/260,384, filed on September 27, 2002,

entitled PROVIDING A CONSISTENT HIERARCHICAL ABSTRACTION OF

20   RELATIONAL DATA, by Nipun Agarwal, Eric Sedlar, Ravi Murthy and Namit Jain;

[0006]   U.S. Patent Application Serial No. 10/259,278, filed on September 27, 2002,

entitled MECHANISM FOR MAPPING XML SCHEMAS TO OBJECT-RELATIONAL

1

DATABASE SYSTEMS, by Ravi Murthy, Muralidhar Krishnaprasad, Sivasankaran

Chandrasekar, Eric Sedlar, Vishu Krishnamurthy and Nipun Agarwal;

[0007]   U.S. Patent Application Serial No. 10/260,161, filed on September 27, 2002,

entitled INDEXING TO EFFICIENTLY MANAGE VERSIONED DATA IN A

5   DATABASE SYSTEM, by Nipun Agarwal, Eric Sedlar and Ravi Murthy;

[0008]   U.S. Patent Application Serial No. 10/256,524, filed on September 27, 2002,

entitled MECHANISMS FOR STORING CONTENT AND PROPERTIES OF

HIERARCHICALLY ORGANIZED RESOURCES, by Ravi Murthy, Eric Sedlar, Nipun

Agarwal, and Neema Jalali;

10   [0009]   U.S. Patent Application Serial No. 10/259,176, filed on September 27, 2002,

entitled MECHANISM FOR UNIFORM ACCESS CONTROL IN A DATABASE

SYSTEM, by Ravi Murthy, Eric Sedlar, Nipun Agarwal, Sam Idicula, and Nicolas Montoya;

[00010]   U.S. Patent Application Serial No. 10/256,777, filed on September 27, 2002,

entitled LOADABLE UNITS FOR LAZY MANIFESTATION OF XML DOCUMENTS by

15   Syam Pannala, Eric Sedlar, Bhushan Khaladkar, Ravi Murthy, Sivasankaran Chandrasekar,

and Nipun Agarwal;

[0010]   U.S. Patent Application Serial No. 10/260,381, filed on September 27, 2002,

entitled MECHANISM TO EFFICIENTLY INDEX STRUCTURED DATA THAT

PROVIDES HIERARCHICAL ACCESS IN A RELATIONAL DATABASE SYSTEM, by

20   Neema Jalali, Eric Sedlar, Nipun Agarwal, and Ravi Murthy;

[0011]   U.S. Patent Application Serial No. **10/648,749**, filed on August 25, 2003, entitled

IN-PLACE EVOLUTION OF XML SCHEMAS, by Sam Idicula, Nipun Agarwal, Ravi

Murthy, and Sivasankaran Chandrasekar (Attorney Docket No. 50277-2237); and

2

[0012]   U.S. Patent Application Serial No. **10/648,497** , filed on August 25, 2003, entitled

MECHANISM TO ENABLE EVOLVING XML SCHEMA, by Sam Idicula, Nipun

Agarwal, Ravi Murthy, Eric Sedlar, and Sivasankaran Chandrasekar (Attorney Docket No.

50277-2238).

5

BACKGROUND AND SUMMARY

[0013]   The present invention relates to database systems, and in particular, to techniques

for directly loading data into a database.

[0014]   Structured data often conforms to a type definition. For example, a type

10   definition for a "person" type may define distinct attributes such as "name," "birthdate,"

"height," "weight," and "gender." Each "instance" of a particular type comprises a separate

value for each of the attributes defined by the particular type. For example, an instance of

the "person" type might comprise values such as "Fred Brown," "January 1, 1980," "72

inches," "240 pounds," and "male." Each attribute is also of a type. For example, the

15   "name" attribute might be of a "string" type, the "birthdate" attribute might be of "date" type,

and the "gender" attribute might be of an "enumerated" type. Structured data might

comprise multiple different instances of the same type.

[0015]   Different approaches may be used to store structured data into a database. One

such approach is called "conventional path loading." According to conventional path

20   loading, a client application parses structured data that comprises one or more instances of a

type. Values within the structured data correspond to attributes of the type. The client

application generates Structured Query Language (SQL) commands, such as INSERT

commands, that, when executed by a database server, cause the database server to insert the

3

values into corresponding columns of a database table. Unfortunately, due to its heavy use of the SQL engine, conventional path loading often suffers in terms of performance and memory consumption.

[0016]   Another approach for storing structured data into a database is called "direct path loading." Through direct path loading, values within structured data are stored directly into a database without going through the SQL engine. By consulting a control file that is associated with the structured data, a client application can determine the types to which instances within the structured data conform. If the structures of the types are defined to the client application, then, based on those structures, the client application can create an array that comprises columns that correspond to the types' attributes. The client application can populate each attribute's corresponding column with values that correspond to that attribute. Once the array is populated, the client application can convert the array into a stream, which the database server can directly convert into the database's data blocks. Direct path loading exhibits performance superior to that of conventional path loading.

[0017]   Some types indicated by a control file may be standard types that are defined to a client application. A scalar type is an example of such a standard type. The client application has information about the characteristics of a scalar type, such as the maximum storage size of a scalar type. With this information, the client can generate the data stream as described above.

[0018]   However, some types indicated by a control file might not be among the types that are defined to the client application. A type indicated by a control file might have a structure that is defined only to a program that implements that type. Although the type

4

might comprise attributes that are of standard types, the control file and the client application might lack any information about the number or types of such attributes.

[0019]   Without such information, the client application cannot generate or populate an array that comprises a separate column for each such attribute. The client application does

5   not possess sufficient information to map values that correspond to such attributes to corresponding columns of a table in a relational database. Consequently, there is no effective way for the client application to store instances of such a type in a database using the direct path loading approach.

[0020]   Types that are not defined to a client application are called "opaque types"

10   relative to the client application, because the internal structure of such types is obscured from the client application. The internal structure of an opaque type, including the number and types of attributes of the opaque type, often are defined only to a program that implements the opaque type. Such a program may be external to both the client application and the database server.

15   [0021]   It may not be practical to modify a client application every time that a new type is introduced, so that the new type is defined to the client application. Additionally, the structures of some existing types may change as time passes. It may be impractical to modify a client application every time that the structure of an existing type changes.

[0022]   One kind of opaque type is an XML type. An example of an XML type is

20   provided in co-pending U.S. Patent Application No. 10/259,278. "XML" stands for "Extensible Markup Language." An XML schema is metadata that describes a hierarchical structure. Instances of the XML schema comprise data that conforms to the structure described by the XML schema. Through XML elements expressed in the structure, an XML

5

schema defines one or more types. XML elements in such a structure may be mapped to columns of database tables. Using the conventional path loading approach, values that correspond to the XML elements may be stored in the columns that are mapped to those XML elements.

5   [0023]   An XML type is special because an XML type may define alternative structures to which instances of the XML type may conform. For example, an XML type definition might indicate that one or more attributes of the XML type are optional. Therefore, if attributes "A," "B," and "C" are optional, then one instance of the XML type might comprise a value for attribute "A," but no values for attributes "B" or "C," while another instance of

10 the XML type might comprise a value for attribute "B," but no values for attributes "A" or "C." Because the instances may conform to alternative defined structures rather than a single defined structure, the instances may be said to comprise "semistructured" data rather than "structured" data.

[0024]   Related application Ser. No. 10/648,577 describes an approach for efficiently

15 performing direct path loading to store opaque data. Related application Ser. No. 10/648,600 describes an approach for efficiently performing direct path loading to store semistructured data.

[0025]   Described herein is a method and mechanism for efficiently loading data into a database using any protocol or client. Examples of such clients/protocols include the File

20 Transfer Protocol (FTP) and Hypertext Transfer Protocol (HTTP). In one embodiment, disclosed is a method and system for storing data into a database, where a determination is made if schema metadata that is used to load the data into the database already exists, and where the existing schema metadata is used to load the data into the database if the schema

metadata already exists. If the appropriate schema metadata does not exist, then it is

generated and cached so that a later load operation for the same schema type will not need to

re-generate this information. In this way, the cost to generate the schema metadata is

amortized over multiple load operations to load data of the same schema type. The approach

5   is protocol neutral so that multiple different protocol-based loads can operate with the same

schema metadata information and load structures.

[0026]   Further details of aspects, objects, and advantages of the invention are described

below in the detailed description, drawings, and claims.

BRIEF DESCRIPTION OF THE DRAWINGS

[0027]   The present invention is illustrated by way of example, and not by way of

limitation, in the figures of the accompanying drawings and in which like reference numerals

refer to similar elements and in which:

5   [0028]   Figure 1A is a flowchart of a process for loading data into a database according to

an embodiment of the invention.

[0029]   Figure 1B illustrates example structures for efficiently loading data into a

database.

[0030]   Figure 1C is a block diagram that illustrates example structures that contain

10   columns for storing values that are not to be displayed to a user;

[0031]   Figure 2 is a block diagram that illustrates a row of a database table in which

multiple values associated with an instance of a child type are stored in-line with values

associated with an instance of a parent type;

[0032]   Figure 3 is a block diagram that illustrates a row of a database table in which

15   multiple values associated with an instance of a child type are stored out-of-line from a

database table that stores values associated with an instance of a parent type;

[0033]   Figure 4A is a block diagram that illustrates multiple rows of a nested database

table in which multiple values associated with an instance of a child type are stored;

[0034]   Figure 4B is a block diagram that illustrates multiple rows of a nested database

20   table in which multiple values associated with an instance of a child type are stored out-of-

line;

8

[0035]   Figure 5 is a block diagram that illustrates a system, according to an embodiment of the present invention, in which semistructured data may be stored in a database according to the direct path loading approach;

[0036]   Figures 6A-6D are flow diagrams that illustrate a technique, according to an embodiment of the present invention, for storing semistructured data in a database according to the direct path loading approach; and

[0037]   Figure 7 is a block diagram that illustrates a computer system upon which an embodiment of the invention may be implemented.

DETAILED DESCRIPTION OF THE INVENTION

[0038]   Described herein is a method and mechanism for efficiently loading data into a

database using any protocol or client. In the following description, for the purposes of

explanation, numerous specific details are set forth in order to provide a thorough

5   understanding of the present invention. It will be apparent, however, that the present

invention may be practiced without these specific details. In other instances, well-known

structures and devices are shown in block diagram form in order to avoid unnecessarily

obscuring the present invention.

10   FUNCTIONAL OVERVIEW

[0039]   It is desirable to store data, such as semistructured data, in a database, e.g.,

according to the direct path loading approach. The present invention can be used to

efficiently load data into a database using any protocol or client, e.g., FTP, HTTP, or

datapump.

15   [0040]   The amount or granularity of data that is loaded by any particular load operation

is highly dependent upon the specific needs of the system, client, or user at any moment in

time. The range of variation in the amount of data to load can be quite significant, with some

operations loading one row of data at a time into a database, while other operations loading

millions of rows of data at a time into the database.

20   [0041]   In some cases, the typical operating parameters or usage characteristics of a given

protocol may relate to the amount or granularity of data that is loaded into a database. For

example, the amount of data that is loaded for each FTP or HTTP related load operation to a

database is typically far less than the amount of data that may be loaded by large-scale native

database-load operations, e.g., for large data warehouse or OLTP archive/load operations.

This is not to say that a FTP or HTTP-related load operation cannot load large quantities of

data in a given operation, but in normal usage, it is not uncommon for a given FTP/HTTP

5   operation to load one or just a few rows of data for each such load operation.

[0042]   The total cost of the operation to load data into the database can be roughly

calculated by taking into account: (1) the preparatory cost to analyze and prepare the system

for the load operation; and (2) the cost to insert and store data in the database. The data can

be inserted into a relational database using SQL or a direct load process as described above.

10   [0043]   The preparatory process to "scan" data and to prepare structures to load the data

into the database consumes an amount of system resources. As described in more detail

below, these costs result from the process of parsing the data to determine the schema and

schema metadata that are associated with the data (e.g., number of columns and column

types) as well as preparing the structures actually used to load the data (e.g., allocating and

15   initializing column arrays). These preparatory costs are relatively constant regardless of the

amount of data to load for any particular operation.

[0044]   The cost to insert and store data are the direct costs associated with populating the

load structures and storing/saving the data into the database. These costs are directly tied to

the amount of data that must be loaded, and will increase as a function of the quantity of data.

20   [0045]   If the load operation for a given operation loads a large amount of data into the

database, then the amount of resources that is needed to identify the schema and prepare the

load structures becomes relatively insignificant relative to the overall process of loading the

data.

[0046]   However, as noted above, certain types of protocol-related load operations have the characteristic of typically using a large number of operations that each load only a small quantity of data per operation. If the amount of data to load for a given operation is small, then the cost of the preparatory process becomes far more significant to the overall load

5   process.

[0047]   According to a first embodiment of the invention, the overall cost of performing the preparatory operations for loading data is amortized over multiple operations, and is therefore reduced per operation, by caching the results of the scan process in the system. Multiple load operations for the same schema type can used the cached data from a prior

10   processing of the schema to load data into the database. In this way, the cost is spread among all of the load operations that use the cached information. In an additional embodiment, the load structures and streams that are created to implement loading of data into the database are not released after a load operation for which they are created; instead, these structures and streams are maintained and re-used for later operations that load data of the same schema

15   type.

[0048]   According to one embodiment of the present invention, to load data for a non-native datatype, such as a type of semistructured data, a program that implements a type to which semistructured data conforms (a "type implementor") registers, with a client application, one or more routines that are associated with the type. The type implementor,

20   which is external to both the client application and a database server that manages the database, implements the routines. In response to the registration, the client application adds an entry to a dispatch table. The entry indicates the association between the type and the routines.

12

[0049]   The client application reads semistructured data that comprises instances of the type. By consulting a control file that is associated with the data, the client application determines that the instances are of the type. The structure of the type, the number of attributes of the type, and the types of those attributes are not defined to the client

5   application. Therefore, the client application locates an entry in the dispatch table that indicates the specified type. The entry indicates the association between the specified type and the routines that are implemented by the type implementor. For example, an entry might indicate an association between an XML type and a set of routines that are implemented by the XML type's implementor.

10   [0050]   The client application invokes the routines. One or more of the routines creates an array for storing instances of the type. The array comprises a separate column for each attribute of the type. One or more of the routines populates the columns with values that are specified in the data. Each such value corresponds to a separate attribute of the type. One or more of the routines stores each such value in a column that corresponds to an attribute that

15   corresponds to that value.

[0051]   The array further may comprise one or more additional columns that correspond to one or more hidden columns of a database table that is to store at least parts of instances of the type. Hidden columns store values that are not displayed to a user when the database table that contains the hidden columns is queried.

20   [0052]   One or more of the routines populates the additional columns with additional values that typically are not specified in the data. For example, for each row of the array, a routine might generate a different instance identifier to distinguish the instance stored in that

13

row from other instances stored in other rows. The routine might populate one of the additional columns with the instance identifiers.

[0053] For another example, for each row of the array, a routine might generate a different positional descriptor that indicates the position of the various attributes within the instance stored in that row. Such positional descriptors are especially useful in relation to instances of an XML type. Although XML documents express data according to a structure in which each attribute value is located at a different position relative to other attribute values, database table rows that store the values of such XML instances typically are not ordered in a way that indicates a position. By populating an additional column with positional descriptors, the original structure of an XML document can be preserved.

[0054] Attributes of one type may conform to other types. For example, a top-level XML type might indicate a first attribute that is of a scalar type, and a second attribute that is of a "purchase order" type. The "purchase order" type also might indicate several attributes. When a type is nested within another type in this manner, routines associated with the nested type are invoked according to the technique described above. For each nested type, the routines associated with that nested type create and populate a separate array that corresponds to that nested type. References to an array generated for a nested type are stored in a corresponding column of an array generated for the type that indicates an attribute that is of the nested type.

[0055] With the arrays populated, the client application converts the arrays, including any additional columns, into a data stream, which the database server understands and converts the data stream to the format of the data blocks in the database. As a result, values that were stored in array columns

14

that correspond to database table hidden columns are stored in those hidden

columns. Note that the SQL engine can be completely bypassed in the whole process.

[0056]   Thus, the client application stores semistructured data in the database according to

the direct path loading approach. Because the direct path loading approach does not require

5   the SQL engine to load each row of data, the direct path loading approach is faster and

consumes less memory than the conventional path loading approach described above.

Because type implementors are external to both the client application and the database server,

instances of new types can be stored in the database without modifying either the client

application or the database server.

10   [0057]   The code that implements the routines that are associated with a particular type

may be centralized within the particular type's implementor rather than being distributed

among multiple separate programmatic components. Such centralization promotes savings in

terms of time and money.

15   REUSE OF STRUCTURES TO LOAD DATA HAVING SIMILAR SCHEMA

[0058]   Figure 1A is a flowchart of a process for efficiently loading data into a database

according to an embodiment of the invention. At 2, identification is made of the data that is

to be loaded into a database. For example, consider if a FTP client wishes to load data into a

database for XML-based data types. The operation to load the XML document into the

20   database is initiated, e.g., using the GET or PUT commands. The GET or PUT operation

will identify the specific XML document to load into the database.

[0059]   At 4, identification is made of the schema for the data to load. In one

embodiment, the schema identifier information can be determined based upon schema

15

identifiers placed in the header or body of the document to load into the database. For example, for an XML-based document, the schema identifier is placed in the document itself.

[0060]   A determination is made whether the preparatory process to scan the data and determine the schema metadata information has already been performed and the results

5   cached in the system (6). The preparatory process might have already been performed, for example, when loading a prior instance of the same XML document type.

[0061]   If the correct schema metadata information is not already cached in the system, then at 8, the schema is described and the appropriate metadata generated. For example, when loading data into a relational or object-relational database, this action will determine

10   the identity, number, and types of columns for the tables into which data will be inserted. In addition, one or more of the columns may contain pointers or refs to other columns. The present action will also identify these other columns that are referenced by the primary columns. In one embodiment, the action of box 8 is referred to as the Describe procedure, and may be implemented as discussed in more detail below. The metadata results are cached

15   so that it can be later re-used when attempting to load another instance of the same schema type.

[0062]   If the Describe information for the data to load already exists in the system, then it is re-used to load the data for the present instance of the data to load (10).

[0063]   A determination is made whether the appropriate structures and streams that

20   should be prepared to load the data into the database have already been prepared, initiated, and still exist in the system (12). Examples of such structures and streams are described with respect to figures 1C-6D and may include such structures as column arrays corresponding to the schema of the data.

[0064]   If such structures and streams already exist for the schema of the present data to

load, then they are identified to be used to load the data (16). If they do not exist, then they

are created (14), e.g., using the processes described with respect to Figures 1C-6D. At 18,

the data can now be loaded into the database.

5   [0065]   Therefore, this process allows data to be loaded into a database very efficiently,

particularly for operations involving many separate loads of small quantities of data. In some

cases, a single instance, e.g., an XMLtype instance, leads to data being loaded into multiple

tables. In one embodiment, multiple streams can be created for loading the data, which can

be parallelized for even faster loads.

10   [0066]   Fig. 1B illustrates some example structures that are employed to efficiently load

data into a database 20 according to the present embodiment of the invention. In this

example configuration, a series of data 40, 42, 44, 46, and 48 are to be loaded into database

20. These data 40, 42, 44, 46, and 48 are received over a network or other data input path 50,

e.g., from a FTP client.

15   [0067]   A loader process 32 handles the function of loading data into database 20. It is

noted that term "process" as used herein may comprise any type of entity such as threads,

tasks, or processes, and is not limited to any particular type of entity.

[0068]   When a first data 40 is identified to be loaded into database 20, a determination is

made regarding the identity of the schema for data 40. This may be performed by loader

20   process 32 or a separate process 34. Assume data 40 is associated with Schema B.

[0069]   The loader process 32 makes a determination whether the Describe information

for Schema B already exists in the system, e.g., at cache 30. If not, then this information is

17

acquired, e.g., by scanning, parsing, and analyzing the data 40 and placing the resulting

metadata information in cache 30 for subsequent loads of data having the same schema type.

[0070]   If they do not yet exist, the appropriate load structures 28 are created for Schema

B. If they already exist, then they are re-used to load data 40 through path 38 into database

5   table(s) 22.

[0071]   The next data 42 to load is another instance of Schema B. Since the schema

information 30 and load structures 28 for Schema B already exist, the data can be loaded

without incurring any additional costs to setup schema information or load structures.

[0072]   The next data 44 to load is an instance of Schema A, which is another schema

10   type. A determination is made regarding the identity of the schema for data 40. The loader

process 32 makes a determination whether the Describe information for Schema A already

exists in the system, e.g., at cache 30. If not, then this information is acquired, e.g., by

scanning, parsing, and analyzing the data 40 and placing the resulting information in cache

30 for subsequent data to load of the same schema type. If they do not yet exist, the

15   appropriate load structures 26 are created for Schema A. If they already exist, then they are

re-used to load data 44 through path 36 into database table(s) 24.

[0073]   The next data 46 to load is another instance of Schema A. Since the schema

information 30 and load structures 26 for Schema A already exist, the data can be loaded

without incurring any additional costs to setup schema information or load structures.

20   **[0074]**   The final data 48 to load is an instance of Schema B. As before, since the schema

information 30 and load structures 28 for Schema B already exist, the data can be loaded

without incurring any additional costs to setup schema information or load structures.

18

[0075]   The cached schema information 30 or load structures 26, 28 and any associated

resources can be released back to the system once they are no longer needed. Alternatively,

they can be timed out and released if not used after a certain period of time or based upon a

least recently used (LRU) approach.

5   [0076]   It is noted that the present embodiment may be used to efficiently load data

corresponding to multiple protocols. For example, in Fig. 1B, assume that data 40 is loaded

from an FTP client while data 42 may be loaded from an HTTP client. The present approach

is protocol neutral since it is keyed upon the schema of the data that is being loaded, and not

upon the protocol of the client that is initiating the load process. Therefore, regardless of the

10   protocol that is being used to load the data, the same cached metadata and load structures can

be used to load data into the database. This allows clients of different protocol types to

concurrently load data into the system without requiring multiple parallel structures to be

created for the same schema, and therefore advantageously leverages the efficiency that is

gained by amortizing the described preparatory procedure over multiple data loads as well as

15   over multiple different protocol-based loads.

ARRAY COLUMNS CORRESPONDING TO HIDDEN COLUMNS

[0077]   Set forth in these sections is an embodiment of an approach for loading

20   semistructured data into a database, such as certain XML-based data. In one embodiment,

one or more routines associated with a particular type generates an array that contains

separate columns for separate attributes of the particular type. The array may also contain

19

one or more columns that correspond to hidden columns of a database table in which instances of the particular type are to be stored.

[0078] Figure 1C is a block diagram that illustrates example structures 100 that contain columns for storing values that are not to be displayed to a user. Structures 100 comprise a

5   database table 102 and an array 104. Database table 102 comprises user-visible columns 106A-106N and hidden columns 108A-108N. User-visible columns 106A-106N correspond to attributes of a type whose instances are to be stored in database table 102.

[0079] Routines associated with a type whose instances are to be stored in database table 102 generate array 104. Array 104 comprises array columns 110A-110N and columns 112A-

10   112N. Columns 110A-110N correspond to user-visible columns 106A-106N. Columns 112A-112N correspond to hidden columns 108A-108N.

[0080] The routines populate columns 110A-110N with values specified in semistructured data. The routines populate columns 112A-112N with additional values, which might not be specified in the semistructured data. For example, the routines may

15   populate column 112A with instance identifiers that distinguish the instances stored in the rows of array 104, and the routines may populate column 112B with positional descriptors that indicate positions of such instances relative to each other within an original document.

[0081] When the routines generate data streams based on array 104, they do so based on the structure of columns 110A-110N and columns 112A-112N and all of the values stored

20   therein. Therefore, when the data are written to a database, values that were stored in columns 110A-110N are stored in user-visible columns 106A-106N, and values that were stored in columns 112A-112N are stored in hidden columns 108A-108N.

[0082]   Hidden columns are useful for storing information that is needed for re-creating

an original document from which values were obtained, especially when that information

cannot be derived from the values themselves. For example, such information may indicate

some structural aspects of an original document. Because such information is typically used

5   by a computer program and not by a human user, hidden columns are especially appropriate

for storing such information.

[0083]   While an embodiment of the invention is described with reference to arrays,

alternative embodiments may use data structures other than arrays to perform the techniques

described herein.

10

OUT-OF-LINE TABLES AND NESTED TABLES

[0084]   As is described above, a type may indicate multiple attributes, one or more of

which may be of another type that indicates other attributes. A type of an attribute indicated

by another type may be called a nested type. For example, if a type "A" indicates an attribute

15   "X" that is of type "B," then "B" is called a nested type. The nested type and the type that

indicates the attribute that is of the nested type may be called the child type and the parent

type, respectively, relative to each other. In the above example, type "A" is a parent type

relative to type "B," and type "B" is a child type relative to type "A."

[0085]   Values within instances that conform to nested types may be stored in database

20   tables in any of a variety of ways. Because values may be stored in database tables in a

variety of ways, techniques for directly loading semistructured data into database tables

should accommodate each of those ways.

[0086]   For example, given a database table that contains multiple columns for multiple attributes of a parent type, multiple values corresponding to multiple attributes of a child type may be stored together within a single column of that database table as a large object (LOB). Internal delimiters within the LOB may delimit the values that correspond to the child type. When values corresponding to a child type are stored in this manner, the values are said to be stored "in-line."

[0087]   Figure 2 is a block diagram that illustrates a row of a database table 200 in which multiple values associated with an instance of a child type are stored in-line with values associated with an instance of a parent type. Values "A," "C," and "D," associated with the parent type, are stored in columns 202A, 202B, and 202D, respectively. Multiple values "(W,X,Y,Z)," associated with the child type, are stored in-line in column 202B as a LOB.

[0088]   Alternatively, multiple values corresponding to multiple attributes of a child type may be stored in a database table that is separate from the database table that contains columns that correspond to attributes of the parent type. A column within the parent type's database table may store references to the child type's database table. Within the child type's database table, the values corresponding to the child type still may be stored in a single column as a LOB. When values corresponding to a child type are stored in this manner, the values are said to be stored "out-of-line."

[0089]   Figure 3 is a block diagram that illustrates a row of a database table 304 in which multiple values associated with an instance of a child type are stored out-of-line from a database table 300 that stores values associated with an instance of a parent type. Values "A," "C," and "D," associated with the parent type, are stored in columns 302A, 302B, and 302D, respectively. Column 302B stores a reference to column 306 of database table 304.

22

Multiple values "(W,X,Y,Z)," associated with the child type, are stored out-of-line in column 306 as a LOB.

[0090]   However, multiple values do not need to be stored as a LOB. Each of the multiple values may be stored in a separate row of a single column in the child type's

5   database table. A first row of the column may store a first of the values, a second row of the column may store a second of the values, and so on. Sets of values within separate instances may be stored in separate sets of rows in the same column of the child type's database table. For example, values of a first instance of the child type may be stored in the first N rows of the column, values of a second instance of the child type may be stored in the next N rows of

10   the column, and so on. When values corresponding to a child type are stored in this manner, the values are said to be stored in a "nested table."

[0091]   Figure 4A is a block diagram that illustrates multiple rows of a nested database table 404 in which multiple values associated with an instance of a child type are stored . Values "A," "C," and "D," associated with the parent type, are stored in columns 402A,

15   402B, and 402D, respectively. Column 402B stores a reference to rows 406A-408D of database table 404. Values "W," "X," "Y," and "Z," associated with the child type, are stored in rows 406A-408D, respectively, of nested table 404. In this case, the values are stored in separate rows, and not as a LOB.

[0092]   Figure 4B is a block diagram that illustrates multiple rows of a nested database

20   table 454 in which multiple values associated with an instance of a child type are stored out-of-line. Values "A," "C," and "D," associated with the parent type, are stored in columns 452A, 452B, and 452D, respectively. Column 402B stores a reference to a table of pointers 458. Each row in table of pointers 458 contains a pointer that points to a separate row of

23

database table 454. Values "W," "X," "Y," and "Z," associated with the child type, are stored in rows 456A-458D, respectively, of nested table 454. In this case also, the values are stored in separate rows, and not as a LOB.

[0093]   To associate the multiple rows in the child type's database table with the single

5   row in the parent type's database table, the corresponding rows from both database tables may be associated with a set identifier that distinguishes those associated rows from other associated rows in the database tables. For example, the first row in the parent type's database table and the first N rows in the child type's database table may be associated with a first unique set identifier, the second row in the parent type's database table and the second N

10   rows in the child type's database table may be associated with a second unique set identifier, and so on.

[0094]   There is no limitation on the levels of indirection that may be used when storing instances of nested types in nested tables. A nested type may indicate an attribute that is of another nested type. Thus, a row of a nested table may indicate a reference to another out-of-

15   line database table or nested table.

[0095]   Based on characteristics of the nested type, one way of storing instances of the nested type may be selected over other ways. A schema processor may make such a selection.

[0096]   A nested table notably is useful for storing a collection that may be defined by an

20   XML schema. Such a collection comprises zero or more instances of a type. The XML schema may indicate a minimum and/or maximum number of instances of the type that are to be contained in the collection. A nested table corresponding to such a collection may be

24

created to comprise a number of rows equal to the maximum number of instances that will be

in the collection.

EXAMPLE SYSTEM FOR DIRECTLY LOADING SEMISTRUCTURED DATA

5   **[0097]**   Figure 5 is a block diagram that illustrates a system 500 in which semistructured

data may be stored in a database according to the direct path loading approach, according to

an embodiment of the present invention. System 500 comprises a client application 502, a

database server 504, a database 506, and type implementors 508A-N. Client application 502,

database server 504, and type implementors 508A-N are coupled communicatively to each

10   other. Database server 504 is coupled communicatively to database 506. In the present

example, client application 502 is loading the data, and can be any type of client entity,

protocol, or process, such as for example, FTP, HTTP, datapump, or any suitable loader type.

[0098]   Client application 502 reads or otherwise receives semistructured data 518 as

input. Semistructured data 518 comprises instances of a type. For example, semistructured

15   data 518 may comprise one or more XML instances that conform to an XML schema.

Semistructured data 518 also comprises an identity of the type. For example, the type may

be identified as an XML type. Semistructured data 518 does not indicate the structure of the

type. The structure of the type is not defined to client application 502.

[0099]   Based on semistructured data 518, client application 502 determines the identity

20   of the type to which the instances conform. Client application consults dispatch table 512 to

find, within the dispatch table, an entry that indicates the identity of the type. Dispatch table

512 comprises a separate entry for each of type implementors 508A-508N. Each entry

indicates memory addresses of routines that are implemented by the type implementor that

25

implements the type that is indicated by that entry. Table 1 below depicts an example of

entries within a dispatch table.

TABLE 1—EXAMPLE DISPATCH TABLE ENTRIES

TYPE IDENTITY	ADDRESSES OF ROUTINES
Identity of type implemented by type implementor 508A	Address of routine 510AA . . . Address of routine 510AN
. . .	. . .
Identity of type implemented by type implementor 508N	Address of routine 510NA . . . Address of routine 510NN

5

[00100]   The entries in dispatch table are added by client application 502 in response to

type implementors 508A-N registering routines 510A-510NN with the client application.

Each type implementor provides client application 502 with the information needed to add an

entry for the type implemented by that type implementor. For example, type implementor

26

508A may load routines 510AA-AN into locations in memory, and then specify those

locations to client application 502.

[00101]   In response to finding, in dispatch table 512, an entry that indicates the identity of

the type, client application 502 invokes the routines located at the memory addresses

5   indicated by the entry. For example, if the type is implemented by type implementor 508A,

then client application 502 invokes routines 510AA-510AN. If the type is an XML type,

then client application 502 invokes routines that are associated with the XML type in the

dispatch table.

[00102]   One or more of the invoked routines creates an array 516A in client application

10   address space 514. Client application address space 514 comprises a segment of memory

allocated for use by client application 502. Array 516A comprises a separate column for

each attribute of the type.

[00103]   For example, an invoked routine may initialize a context for the type. Another

invoked routine may indicate a number of columns in a database table that is to store

15   instances of the type. For example, one implementation of an XML type maps to two top-

level columns of a database table, in which one of the two top-level columns is a hidden

column. The hidden column is for storing a positional descriptor, and the user-visible

column is for storing a LOB or a reference to an out-of-line or nested table. Therefore, an

invoked routine of the XML type implementor creates an array with two columns that

20   correspond to the two top-level columns of the database table.

[00104]   One or more of the invoked routines populates the columns of array 516A,

including columns that correspond to hidden columns of a database table, with values. Some

values may be specified in semistructured data 518, and other values may be derived from

27

values specified in the semistructured data. Each value specified in semistructured data 518 corresponds to a separate attribute of the type.

[00105]   Client application 502 may pass such values as parameters to one or more of the invoked routines. For example, for each instance specified within semistructured data 518, client application 502 may pass a single block of combined values, which represent that instance, as a parameter to one or more routines. The one or more routines may parse the block of combined values to produce separate values that correspond to the separate attributes of the type.

[00106]   One or more of the invoked routines stores the values in corresponding columns of array 516A. Each row of array 516A stores values for a different instance specified in semistructured data 518. One of more the invoked routines returns, to client application 502, one or more pointers to one or more addresses within client application address space 514 at which one or more populated rows of array 516A can be found. Using the one or more pointers, client application 502 can locate and read the populated rows of array 516A.

[00107]   One or more of the attributes of the type may be of a nested type that indicates one or more other attributes. For example, a parent type might indicate a first attribute that is of a scalar type, and a second attribute that is of a "purchase order" child type. The "purchase order" type also might indicate several attributes. In this case, the routines associated with the parent type invoke routines associated with the child type. The client application does not need to be aware of or invoke routines associated with the child type. When a routine is invoked to describe the structure of the parent type, that routine invokes another routine to describe the structure of the child type. When a routine is invoked to create an array for the parent type, that routine invokes another routine to create an array for

28

the child type. When a routine is invoked to populate the array for the parent type, that

routine invokes another routine to populate the array for the child type.

[00108]   The number of columns in an array generated by a child type's implementor is

based on the number of columns in a database table that is to store instances of the child type.

5   If instances of a child type are stored in a nested table, then the array generated for the child

type may contain one column that corresponds to the one column of the nested table, and

each instance of the child type may be stored in a different row of the array column. Thus,

multiple rows of a child array may correspond to a single row of a parent array. To preserve

this correspondence, a set identifier is generated. The set identifier links the row of the

10   parent array with the corresponding rows of the child array. The set identifier is stored in the

parent array.

[00109]   Each array may be populated independently of each other array. Arrays may be

loaded and streamed asynchronously and independently.

[00110]   Based on the populated rows of one or more arrays, the client application 502

15   generates a data stream. A data stream for one array may be generated independently of a

data stream for another array. The data stream conforms to the format of data blocks within

database 506. As a result, the data stream generated by client application 502 may be written

directly to database 506 without causing the SQL engine to load each row of data. Client

application 502 streams the data to database server 504. A stream generated based on one

20   array may be sent to the database server independently of a stream generated based on

another array. Database server 504 writes the data received from client application 502

directly into one or more data blocks in database 506. Values in array columns that

correspond to hidden columns in database tables are stored in the corresponding hidden columns as a result of the writing.

MEMORY MANAGEMENT

5   **[00111]**   Because the amount of memory available in client application address space 514 is limited, array 516A might not comprise enough rows to store, concurrently, values of all of the instances that are specified in semistructured data 518. Therefore, after a specified number of rows of array 516A have been populated, a data stream may be generated based on those populated rows, and the data may be streamed to database server 504. Then, the

10   memory that the populated rows occupy may be freed. Once the memory has been freed, array 516A may be re-populated with values of additional instances for which a data stream has not yet been generated. This process may be repeated until data streams for all of the instances indicated in semistructured data 518 have been generated and streamed to database server 504.

15   **[00112]**   Different arrays may be populated independently of each other. Different arrays may be associated with different database tables that comprise different numbers of rows. Values stored in such different arrays may be flushed to persistent storage at different times. Memory that stores one array may be freed before other memory that stores another array. For example, a parent array may be associated with a database table that comprises five rows,

20   while a child array may be associated with a nested database table that comprises twenty-five rows. Rows in the child array may be populated, streamed to the database server, and re-populated multiple times before rows in the parent array are streamed to the database server once.

MANAGING NESTED TYPES

[00113]   Semistructured data 518 might comprise instances of a type "A" that comprises

two attributes: an attribute "B" of a scalar type and an attribute "C" of a type "D" that is

5   implemented by type implementor 508A. In turn, type "D" might comprise two attributes

"E" and "F," both of scalar types.

[00114]   In this example, the structure of type "A" is defined, at a high level, to client

application 502. Client application 502 possesses sufficient information to generate, in client

application address space 514, an array (array "A") to store instances of type "A". Array "A"

10   comprises a column (column "B") for attribute "B" and a column (column "C") for attribute

"C."

[00115]   Client application 502 populates rows of array "A" on a per-instance basis.

Because attribute "B" is of a scalar type, client application 502 may populate column "B"

without invoking any external routines. Because attribute "C" is of a type implemented by

15   type implementor 508A, client application 502 invokes one or more of routines 510AA-AN

for each instance of type "A." For each instance, client application 502 passes a combined

value block, which represents the value of attribute "C" for that instance, to the routines.

[00116]   For each instance, the routines generate, in client application address space 514,

columns (columns "E" and "F") corresponding to attributes "E" and "F" of type "D." For

20   each instance, the routines populate columns "E" and "F" with corresponding values of

attributes "E" and "F" separated out from the combined value block received as a parameter.

For each instance, the routines return, to client application 502, a pointer to populated

31

columns "E" and "F". Client application 502 stores the pointer in column "C" in the

instance's corresponding row of array "A."

[00117]   Client application 502 does not possess sufficient information about type "D" to

free memory that stores instances of type "D." Therefore, when client application 502 is

5   going to free memory that stores values for attribute "B," the client application also invokes

one or more of routines 510AA-AN to free memory that stores the corresponding values for

attributes "E" and "F."

[00118]   When client application 502 is going to free memory that contains a pointer to

columns "E" and "F", the client application passes the pointer as a parameter to one or more

10   of routines 510AA-AN. Those routines then free the memory to which the pointer points.

Client application 502 does not free the memory that stores the pointer until the client

application invokes the routines that free the memory to which the pointer points.

REFERENCE COUNTS

15   [00119]   Multiple arrays may reference the same temporary data structure in memory.

Such a data structure should be maintained in memory as long as at least one array is

referencing the data structure. Only after no arrays are referencing a data structure should the

memory that stores that data structure be freed for other purposes. Therefore, a separate

reference count may be associated with each such temporary data structure in memory.

20   When an array begins to make reference to a given data structure, then the reference count

associated with that data structure is incremented. When an array no longer needs to make

reference to a given data structure, then the reference count associated with that data

structure is decremented. When a data structure's reference count is zero, then the memory

that stores that data structure may be freed for other purposes.

EXAMPLE ROUTINES IMPLEMENTED BY A TYPE IMPLEMENTOR

5   **[00120]**   Client application 502 may invoke different routines to perform different

functions relative to a type. For example, each of routines 510AA-AN may perform a

different function relative to a type implemented by type implementor 508. Such functions

may include:

- Allocating and initializing a general context block;

10 - Indicating a type of a database table in which instances of the type are to be stored;

- Indicating a number of columns in a database table in which instances of the type are

   to be stored;

- Indicating types of columns in a database table in which instances of the type are to

   be stored;

15 - Initializing a context for a database table in which instances of the type are to be

   stored;

- Allocating memory, in client application address space 514, for one or more arrays to

   store values of instances of the type;

- Parsing combined value blocks and storing separated values into separate columns of

20 one or more arrays; .

- Freeing memory that stores values of an instance of a type;

- Flushing existing populated memory structures to persistent storage;

- Completing the direct path loading and freeing all memory that was allocated to perform the direct path loading; and

- Aborting the direct path loading and freeing all memory that was allocated to perform the direct path loading.

5   [00121]   Different routines may accept different parameters and return different results. Client application 502 may invoke one or more of routines 510AA-510AN sequentially in an order designed to achieve the ultimate goal of storing instances of the type in database 506 according to the direct path loading approach.

10   ERROR MANAGEMENT

[00122]   Client application 502 may specify actions to be performed when an error occurs during the performance of any of the techniques described herein. For example, when such an error occurs, client application 502 may update an error counter value and determine whether the error counter value is greater than a specified threshold. Client application 502

15   may indicate that the techniques currently being performed should continue, despite the errors, unless the error counter value is greater than the specified threshold. Client application 502 may indicate that the techniques currently being performed, and techniques that will be performed thereafter, should be aborted if the error counter value is greater than the specified threshold.

20   [00123]   It is desirable for routines 510AA-NN to handle errors in a way that is consistent with the way that client application 502 handles errors. Therefore, according to one embodiment, client application 502 passes memory addresses of error handling routines, which are implemented by the client application, to routines 510AA-NN. When any of

routines 510AA-NN determines that an error has occurred, that routine executes an error

handling routine that is located at the specified memory address. Thus, if an error handling

condition is satisfied during the execution of any of routines 510AA-NN, actions connected

to the satisfaction of the condition will be performed just as if the condition had been

5   satisfied outside of the execution of such a routine.

EXAMPLE TECHNIQUE FOR STORING SEMISTRUCTURED DATA IN A

DATABASE ACCORDING TO THE DIRECT PATH LOADING APPROACH

[00124]   Figures 6A-6D are flow diagrams that illustrate a technique 600, according to an

10   embodiment of the present invention, for storing semistructured data in a database according

to the direct path loading approach. Such semistructured data may comprise multiple

different instances that conform to an XML schema.

[00125]   In block 602, a "first" type implementor registers, with a client application,

routines that are implemented by the first type implementor (the "first routines"). For

15   example, type implementor 508A may register routines 510AA-AN with client application

502.

[00126]   In block 604, the client application adds, to a dispatch table, an entry that

indicates an association between the first routines and a parent type that the first type

implementor implements. For example, client application 502 may add, to dispatch table

20   512, an entry that indicates an association between routines 510AA-AN and the type

implemented by type implementor 508A. The parent type includes an attribute that is of a

child type that is implemented by a "second" type implementor.

[00127]   In block 606, the second type implementor registers, with the client application, routines that are implemented by the second type implementor (the "second routines"). For example, type implementor 508B may register routines 510BA-BN with client application 502.

[00128]   In block 608, the client application adds, to the dispatch table, an entry that indicates an association between the second routines and the child type that the second type implementor implements. For example, client application 502 may add, to dispatch table 512, an entry that indicates an association between routines 510BA-BN and the type implemented by type implementor 508B.

[00129]   In block 610, the client application receives semistructured data that specifies instances of the parent type. Each instance of the parent type specifies an instance of the child type. For example, client application 502 may read semistructured data 518, which may indicate (1) values of instances of the parent type and (2) the identity of the parent type.

[00130]   In block 612, the client application determines, from the dispatch table, which routines are associated with the parent type. For example, client application 502 may determine, from dispatch table 512, that the parent type is associated with the first routines, routines 510AA-AN.

[00131]   In block 614, the client invokes one or more of the first routines, which are associated with the parent type. For example, client application 502 may invoke routine 510AA to initialize a context block. Client application 502 may invoke routine 510AB to determine a type of a database table that will store instances of the parent type (the "parent database table"). Client application 502 may invoke routine 510AC to determine how many

columns are in the parent database table. Client application 502 may invoke routine 510AD

to determine the types of the columns in the parent database table.

[00132] In block 616, in response to its invocation, one or more of the first routines

creates a "parent" array that comprises (1) a separate column for each attribute of the parent

5   type and (2) columns that correspond to hidden columns of the parent database table. For

example, when invoked, routine 510AE may create array 516A in client application address

space 514. Array 516A might comprise two columns, one of which corresponds to a hidden

column of a database table.

[00133] In block 622, one or more of the first routines invokes one or more of the second

10   routines, which are associated with the child type. For example, routine 510AA may invoke

routine 510BA to initialize a context block. Routine 510AB may invoke routine 510BB to

determine a type of a database table that will store instances of the child type (the "child

database table"). Routine 510AC may invoke routine 510BC to determine how many

columns are in the child database table. Routine 510AD may invoke routine 510BD to

15   determine the types of the columns in the child database table.

[00134] In block 624, in response to its invocation, one or more of the second routines

creates a "child" array that comprises (1) a separate column for each attribute of the child

type and (2) columns that correspond to hidden columns of the child database table. For

example, when invoked, routine 510BE may create array 516B in client application address

20   space 514. Array 516B might comprise five columns, two of which correspond to hidden

columns of a database table.

[00135] Blocks 626-628 may be performed concurrently with blocks 630-632. In block

626, in response to its invocation, one or more of the first routines populates, with values of

instances of the parent type, parent array columns that correspond to the attributes of the parent type. For example, when invoked, routine 510AF may populate one or more columns of array 516A with values of instances that are specified in semistructured data 518.

[00136]   In block 628, in response to its invocation, one or more of the first routines

5   populates, with other values, parent array columns that correspond to hidden columns of the parent database table. The other values might not be specified in the semistructured data, and might not be values that correspond to attributes of the parent type. However, the other values might be derived from the semistructured data. For example, when invoked, routine 510AF may populate some of the columns of array 516A with other values that are derived

10   from semistructured data 518.

[00137]   In block 630, in response to its invocation, one or more of the second routines populates, with values of instances of the child type, child array columns that correspond to the attributes of the child type. For example, when invoked, routine 510BF may populate one or more columns of array 516B with child type instance values that are specified in

15   semistructured data 518.

[00138]   In block 632, in response to its invocation, one or more of the second routines populates, with other values, child array columns that correspond to hidden columns of the child database table. The other values might not be specified in the semistructured data, and might not be values that correspond to attributes of the child type. However, the other values

20   might be derived from the semistructured data. For example, when invoked, routine 510BF may populate some of the columns of array 516B with other values that are derived from semistructured data 518.

38

[00139]   One or more set identifiers may be included among the values with which the

parent and child arrays are populated. As described above, a set identifier associates a row in

the parent array with a set of rows in a child array. Both the parent and child arrays may be

generated with a column to store a set identifier.

5   [00140]   In block 634, the client application generates a data stream based on the populated

rows of one or more of the arrays. For example, based on populated arrays 516A and 516B,

client application 502 may generate one or more data streams that conform to the format of

data blocks stored in database 506.

[00141]   In block 636, the client application streams the data to a database server 504. For

10   example, client application 502 may stream data to database server 504.

[00142]   In block 638, the database server writes the data directly into the database. For

example, database server 504 may write data received from client application 502 directly

into one or more data blocks in database 506. Values in array columns that correspond to

hidden columns in database tables are stored in the corresponding hidden columns as a result

15   of the writing.

[00143]   Thus, semistructured data, such as instances of an XML schema, may be stored in

a database according to the direct path loading approach. As is discussed above, the direct

path loading approach is faster and consumes less memory than the conventional path

loading approach. Using the techniques and systems described above, instances of an XML

20   type may be stored in a database even after the XML type definition has been extended,

without modifying either the client application or the database server.

SYSTEM ARCHITECTURE OVERVIEW

[00144]   The execution of the sequences of instructions required to practice the invention

may be performed in embodiments of the invention by a computer system 1400 as shown in

Fig. 7. As used herein, the term computer system 1400 is broadly used to describe any

5   computing device that can store and independently run one or more programs. In an

embodiment of the invention, execution of the sequences of instructions required to practice

the invention is performed by a single computer system 1400. According to other

embodiments of the invention, two or more computer systems 1400 coupled by a

communication link 1415 may perform the sequence of instructions required to practice the

10   invention in coordination with one another. In order to avoid needlessly obscuring the

invention, a description of only one computer system 1400 will be presented below;

however, it should be understood that any number of computer systems 1400 may be

employed to practice the invention.

[00145]   Each computer system 1400 may include a communication interface 1414

15   coupled to the bus 1406. The communication interface 1414 provides two-way

communication between computer systems 1400. The communication interface 1414 of a

respective computer system 1400 transmits and receives signals, e.g., electrical,

electromagnetic or optical signals, that include data streams representing various types of

information, e.g., instructions, messages and data. A communication link 1415 links one

20   computer system 1400 with another computer system 1400. A computer system 1400 may

transmit and receive messages, data, and instructions, including program, i.e., application,

code, through its respective communication link 1415 and communication interface 1414.

Received program code may be executed by the respective processor(s) 1407 as it is

40

received, and/or stored in the storage device 1410, or other associated non-volatile media, for later execution.

[00146]   In an embodiment, the computer system 1400 operates in conjunction with a data storage system 1431, e.g., a data storage system 1431 that contains a database 1432 that is

5   readily accessible by the computer system 1400. The computer system 1400 communicates with the data storage system 1431 through a data interface 1433. A data interface 1433, which is coupled to the bus 1406, transmits and receives signals, e.g., electrical, electromagnetic or optical signals, that include data streams representing various types of signal information, e.g., instructions, messages and data. In embodiments of the invention,

10   the functions of the data interface 1433 may be performed by the communication interface 1414.

[00147]   Computer system 1400 includes a bus 1406 or other communication mechanism for communicating instructions, messages and data, collectively, information, and one or more processors 1407 coupled with the bus 1406 for processing information. Computer

15   system 1400 also includes a main memory 1408, such as a random access memory (RAM) or other dynamic storage device, coupled to the bus 1406 for storing dynamic data and instructions to be executed by the processor(s) 1407. The main memory 1408 also may be used for storing temporary data, i.e., variables, or other intermediate information during execution of instructions by the processor(s) 1407. The computer system 1400 may further

20   include a read only memory (ROM) 1409 or other static storage device coupled to the bus 1406 for storing static data and instructions for the processor(s) 1407. A storage device 1410, such as a magnetic disk or optical disk, may also be provided and coupled to the bus 1406 for storing data and instructions for the processor(s) 1407. A computer system 1400

41

may be coupled via the bus 1406 to a display device 1411, such as, but not limited to, a

cathode ray tube (CRT), for displaying information to a user. An input device 1412, e.g.,

alphanumeric and other keys, is coupled to the bus 1406 for communicating information and

command selections to the processor(s) 1407.

5   [00148]   According to one embodiment of the invention, an individual computer system

1400 performs specific operations by their respective processor(s) 1407 executing one or

more sequences of one or more instructions contained in the main memory 1408. Such

instructions may be read into the main memory 1408 from another computer-usable medium,

such as the ROM 1409 or the storage device 1410. Execution of the sequences of

10   instructions contained in the main memory 1408 causes the processor(s) 1407 to perform the

processes described herein. In alternative embodiments, hard-wired circuitry may be used in

place of or in combination with software instructions to implement the invention. Thus,

embodiments of the invention are not limited to any specific combination of hardware

circuitry and/or software.

15   [00149]   The term "computer-usable medium" or "computer-readable medium" as used

herein, refers to any medium that provides information or is usable by the processor(s) 1407.

Such a medium may take many forms, including, but not limited to, non-volatile, volatile and

transmission media. Non-volatile media, i.e., media that can retain information in the

absence of power, includes the ROM 1409, CD ROM, magnetic tape, and magnetic discs.

20   Volatile media, i.e., media that can not retain information in the absence of power, includes

the main memory 1408. Transmission media includes coaxial cables, copper wire and fiber

optics, including the wires that comprise the bus 1406. Transmission media can also take the

form of carrier waves; i.e., electromagnetic waves that can be modulated, as in frequency,

42

amplitude or phase, to transmit information signals. Additionally, transmission media can take the form of acoustic or light waves, such as those generated during radio wave and infrared data communications.

5   **[00150]**   In the foregoing specification, the invention has been described with reference to specific embodiments thereof. It will, however, be evident that various modifications and changes may be made thereto without departing from the broader spirit and scope of the invention. For example, the reader is to understand that the specific ordering and combination of process actions shown in the process flow diagrams described herein is

10   merely illustrative, and the invention can be performed using different or additional process actions, or a different combination or ordering of process actions. The specification and drawings are, accordingly, to be regarded in an illustrative rather than restrictive sense.